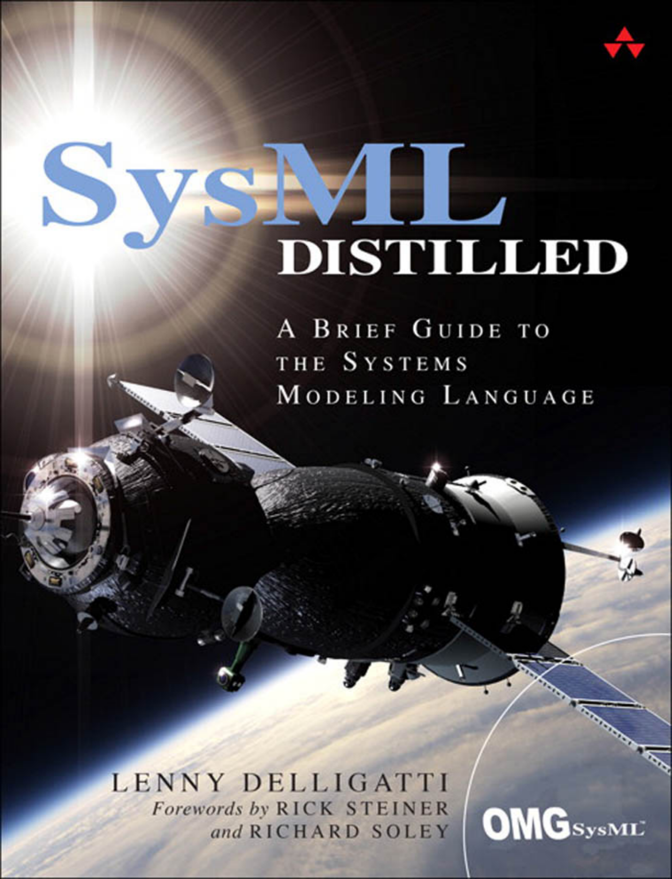

SysML Distilled

A Brief Guide to the Systems Modeling Language

Lenny Delligatti

Forewords by Rick Steiner
and Richard Soley

OMG SysML™

Chapter 1

Overview of Model-Based Systems Engineering

MBSE is the formalized application of modeling to support system requirements, design, analysis, verification, and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases.

—INCOSE, *Systems Engineering Vision 2020*

You're reading this book to learn the Systems Modeling Language (SysML)—either to create SysML models on your systems engineering team, to earn the OMG (Object Management Group) Certified Systems Modeling Professional (OCSMP) certification, or both. SysML, however, is only one facet of a larger subject: model-based systems engineering (MBSE). MBSE is a practice; it's something you *do*. And SysML is a graphical modeling language that enables you to practice MBSE. The practice of MBSE provides the context—and the business case—for learning SysML.

I begin this chapter by answering the basic question, What is MBSE? I then discuss the three pillars of MBSE—the three enablers for the practice of model-based systems engineering. I end by making you aware of a common myth that has arisen about MBSE so that you can better manage your customers' expectations while you deliver the return on investment that MBSE promises.

1.1 What Is MBSE?

The best way to understand the MBSE approach is to begin by understanding the alternative: what modeling practitioners call the **document-based approach** to engineering—and what nonpractitioners call "the way we've always done things." Whether they apply the document-based approach or MBSE, systems engineers perform the same life cycle activities described in the *INCOSE Systems Engineering Handbook* (INCOSE stands for International Council on Systems Engineering). The key difference between the two approaches, however, is the nature of the primary artifacts produced in those life cycle activities.

With the document-based approach, systems engineers manually generate some subset of the following artifacts: concept of operations (ConOps) documents, requirements specifications, requirement traceability and verification matrices (RTVMs), interface definition documents (IDDs), N^2 charts (also known as N-squared charts—matrices of structural interfaces), architecture description documents (ADDs), system design specifications, test case specifications, and specialty engineering analyses (e.g., analyses of reliability, availability, schedulability, throughput, and response time). Document-based systems engineers produce these artifacts in the form of a disjoint set of text documents, spreadsheets, diagrams, and presentations (and configuration-manage them in a disjoint set of repositories).

The problem is this: The document-based approach to systems engineering is expensive. More precisely, it's more expensive than it needs to be; you incur a significant percentage of total life cycle cost maintaining that disjoint set of artifacts. And if you don't pay that cost, the artifacts become inconsistent and obsolete.

Consider the following day-in-the-life scenario. A system architect makes a fourth-iteration design decision to refactor a single block in the system hierarchy into two blocks to achieve a better separation of concerns. She decides to rename the original block to better convey its new, narrower focus. To implement this change completely and consistently, she needs to locate every text document, table, matrix, diagram, and presentation that contains the block, open each one sequentially from the various file servers, intranet websites, and configuration management (CM) repositories where it resides, and then manually type the same change into all of those artifacts.

This approach is time consuming and error prone. For one thing, the architect may mistype the new block name. More important, she needs to know ahead of time which artifacts will be impacted, perhaps a great many. She will likely miss a handful of them, and they will become inconsistent with the rest of the set. That creates problems for the development teams who rely on them as inputs for their stage of the life cycle. It's also a problem for the project manager, who must account for the schedule slippage and increased life cycle cost to fix any defects that propagate down the line.

This scenario is commonplace in organizations that practice the traditional document-based approach to systems engineering. Inconsistency is the problem. And MBSE—when practiced correctly—is the solution.

With the MBSE approach, systems engineers perform the same life cycle activities and produce the same set of deliverables. But the deliverables are not the immediate outputs of the life cycle activities; they are not the primary artifacts. With the MBSE approach, the primary artifact of those activities is an integrated, coherent, and consistent **system model,** created by using a dedicated systems modeling tool. All other artifacts are secondary—automatically generated from the system model using that same modeling tool.

The system model serves as a central repository for design decisions; each design decision is captured as a **model element** (or a relationship between elements) in a single place within the system model. With the MBSE approach, all diagrams and autogenerated text artifacts are merely *views* of the underlying system model; they are not the model itself. And that distinction is the root of the return on investment (ROI) that MBSE offers over the traditional approach.

Let's return to our day-in-the-life scenario, this time with the MBSE approach. The system architect decides to rename the original (refactored) block to better convey its new, narrower focus. To implement this change completely and consistently, she locates that one block within the system model hierarchy (often by using a keyword search in the modeling tool) and types the new name for that block one time. That's it.

The modeling tool automatically (and instantly) propagates the change to all diagrams where that block appears, no matter how large the set may be. The diagrams, after all, are merely views of the underlying model. If that model changes, the diagrams change. The modeling tool also inserts the change into all the autogenerated text artifacts the next time the architect exports them from the model. There is no opportunity for inconsistency to occur between the various views of the model.

MBSE promises increased quality and affordability for one simple reason: The cheapest defect to fix is the one you prevented. And at the heart of this approach is this new kind of engineering artifact called the system model.

The approach that I've described is actually a hybrid form that bridges the gap between MBSE and the document-based approach. Model-based engineering organizations are constrained to adopt this hybrid approach when their customer requires text artifacts as deliverables for review and approval. Organizations that do not have this constraint, however, can practice MBSE in its pure form.

Organizations that attain the highest MBSE maturity level forgo the creation of text artifacts entirely. The system model itself is the artifact that gets reviewed and approved. It's the artifact that gets handed off, refined, and evolved as it goes from one stage of design to the next.

Organizations that produce software systems can even use a (sufficiently robust) modeling tool to transform a system model into a software model and, ultimately, into production-quality source code. This level of MBSE maturity blurs the line between design and development, enabling rapid prototyping and system simulation. At all times, however, the model remains the primary artifact that gets modified when customer requirements change and new design decisions are made. All other artifacts, including source code, are autogenerated byproducts of the model, continuously consistent with the model and with each other.

This is MBSE.

1.2 The Three Pillars of MBSE

How do you do MBSE? What do you need to know?

In short, you need to know three things: a modeling language, a modeling method, and a modeling tool. I refer to these as the three pillars of MBSE. As a member of a design team that's creating an integrated system model, you will use a dedicated modeling tool to perform a set of design tasks prescribed by a modeling method to add elements (and relationships between elements) to an integrated system model that is expressed in a standard modeling language.

Knowledge of a modeling language alone enables you to sketch system design ideas on paper or a whiteboard to quickly and effectively communicate with other team members. Learning a modeling language is the first skill you should acquire, and it's the focus of this book. However, the practice of MBSE requires you to possess all three skills to gain the ROI that this approach offers.

Let's examine each pillar in more detail.

1.2.1 Modeling Languages

When you create a model, you are speaking a language. It's not the natural language you learned as a child at home and in school. It's not the natural language I'm using to communicate with you right now. Rather, it's a **modeling language:** a semiformal language that defines the kinds of elements you're allowed to put into your model, the allowable relationships between them, and—in the case of a graphical modeling language—the set of notations you can use to display the elements and relationships on diagrams.

MBSE practitioners commonly use the **Systems Modeling Language (SysML)** to construct models of a system's structure, behavior, requirements, and constraints. SysML is the focus of this book, but it's not the only modeling language. Engineers and analysts in other design domains (e.g., systems-of-systems, software, hardware, performance, business processes) have other modeling languages available that are more appropriate for the types of systems they design. Like SysML, some of those languages are graphical modeling languages (e.g., UML, UPDM, BPMN, MARTE, SoaML, IDEFx); others are text modeling languages (e.g., Verilog, Modelica).

The key idea here is that each modeling language is a standardized medium for communication; the rules defined in a given language give the model's elements and relationships unambiguous meaning. The capability to construct and read well-formed models is at the heart of the MBSE approach.

1.2.2 Modeling Methods

Learning a modeling language is only the first step on the MBSE path. A modeling language defines a **grammar:** a set of rules that determines whether a given model is well formed or ill formed. Those rules do not dictate how and when to use the language to create a model; they stop short of dictating any particular modeling method.

In contrast, a **modeling method** is something like a road map; it's a documented set of design tasks that a modeling team performs to create a system model. More precisely, it's a documented set of design tasks that ensures that everyone on the team is building the system model consistently and working toward a common end point. Without such guidance, there will be wide variance in the breadth, depth, and fidelity that each member of the team builds into the system model.

Like all projects, an MBSE project requires a plan. And every plan begins with a purpose. Your team will begin by answering the following questions: Why are you modeling? More precisely, what are the expected results of the modeling effort? Are you creating a model only to serve as the central record of authority for all design decisions? Do you need to autogenerate text artifacts from the model for review and approval? Will you use the model to manage requirements traceability and perform downstream impact analysis? Will you use the model to perform trade studies of alternative configurations? Will the system model be integrated with dedicated equation-solving tools and simulation tools to execute the model directly? Will the model itself be an input for the work of downstream design and development teams, such as software, hardware, reliability/availability/performance analysis? Will the model contain the integration and acceptance test cases that will verify system assembly after development? The answers to these questions determine the **purpose** of your team's modeling effort.

Once your team has defined that purpose, you can then answer a new set of questions. How much of the external environment of your proposed system needs to be modeled? Which parts of your system need to be modeled? Which behaviors need to be modeled? How deeply do you need to decompose the internal structures and behaviors? Which details need to be in the model, and which details can be omitted (and left to the discretion of the development teams at implementation time)? The answers to these questions determine the **scope** of the system model your team needs to build.

The definition of scope sets the goalpost that your team is working toward; it enables your team to determine when the model is complete. To be clear, your team will evolve the model over time as requirements change and new design decisions are made. "Complete" in this context means that the model satisfies the purpose you outlined in the project plan.

The scope of the model also determines the modeling method that your team will follow. Several modeling methods are documented in the literature. Your team can adopt one of those existing methods and tailor it to meet your needs and objectives. Or you can create a custom modeling method if none of the existing ones is a good fit. That discussion, however, is beyond the scope of this book.

The focus here is to help you become proficient in SysML, a modeling language, and not to teach you any particular modeling method. SysML is method independent; you can use SysML to create a system model no matter which modeling method you decide is the best fit for your needs. However, I use a little real estate here to list some well-known modeling methods (and some references that provide in-depth coverage of them) to help you on your journey:

- **Method:** INCOSE Object-Oriented Systems Engineering Method (OOSEM)

  Reference: Friedenthal, Sanford, et al., *A Practical Guide to SysML, Second Edition: The Systems Modeling Language* (Boston: MK/OMG Press, 2011)
- **Method:** Weilkiens System Modeling (SYSMOD) method

  Reference: Weilkiens, Tim, *Systems Engineering with SysML/UML: Modeling, Analysis, Design* (Boston: MK/OMG Press, 2008)
- **Method:** IBM Telelogic Harmony-SE

  Reference: Hoffmann, Hans-Peter, "Harmony-SE/SysML Deskbook: Model-Based Systems Engineering with Rhapsody," Rev. 1.51, Telelogic/I-Logix white paper (Telelogic AB, May 2006)

These modeling methods broadly span many stages of the systems engineering life cycle. Not every step prescribed by these methods will apply to your project. Any modeling method you adopt needs to be tailored to meet your project's specific needs. These methods are good starting points.

1.2.3 Modeling Tools

Developing proficiency with a modeling tool is the third pillar of MBSE. **Modeling tools** are a special class of tools that are designed and implemented to comply with the rules of one or more modeling languages, enabling you to construct well-formed models in those languages.

Modeling tools are distinct from diagramming tools such as Visio, Schematic, SmartDraw, ProcessOn, and others. With a **diagramming tool,** you create diagrams—shapes on a page. There is no model underlying those diagrams that ensures automated consistency between them. In contrast, with a modeling tool, you create a model—a set of elements and relationships between elements, and optionally a set of diagrams that serve as views of the underlying model.

When you modify an element on a diagram within a modeling tool, you're actually modifying the element itself in the underlying model. The modeling tool then instantaneously updates all the other diagrams that display that same element. This is a powerful capability—and one that is offered only by this distinct class of tools.

Note that a modeling language specification, such as SysML, is vendor neutral. A particular modeling tool is one vendor's implementation of that language specification. Several commercial tool vendors and nonprofit consortiums have created modeling tools for the various modeling languages. These tools vary in cost, capability, and compliance with the modeling language specifications. Selecting the best tool—based on your project's specific needs and cost constraints—should be part of the MBSE adoption process in your organization.

Much like SysML and the other modeling languages, I am vendor neutral. I do not hawk one product over another in this book. However, I list some SysML modeling tools for you to research for the trade study that your organization will inevitably conduct. The following are commercial-grade (euphemism for "not free") modeling tools:

- Agilian (vendor: Visual Paradigm)
- Artisan Studio (vendor: Atego)
- Enterprise Architect (vendor: Sparx Systems)
- Cameo Systems Modeler (vendor: No Magic)
- Rhapsody (vendor: IBM Rational)
- UModel (vendor: Altova)

The following are free modeling tools, offered with an Eclipse Public License (EPL) or General Public License (GPL):

- Modelio (creator: Modeliosoft)
- Papyrus (creator: Atos Origin)

There are many factors to consider when you select a tool. However, I strongly recommend that you select a tool that is XML Metadata

Interchange (XMI) compliant. The XMI standard enables compliant tools to exchange model data. This capability will ensure that your team avoids the vendor lock-in trap when your needs (and cost constraints) change in the future.

1.3 The Myth of MBSE

The myth of MBSE arises among **stakeholders**—external customers and internal downstream design and development teams—who know *about* MBSE but don't practice it themselves. These are the stakeholders who expect deliverables from you. They understand—conceptually, at least—that you will autogenerate those deliverables from the system model.

The myth they harbor deep inside is that MBSE is an Easy Button: You push it, and good things pop out. To put this more concretely, they believe, incorrectly, that MBSE makes every engineering task easier and reduces cost at every point in the life cycle.

But in truth, MBSE does not (and cannot) eliminate the difficult work of architecting and designing a system well. It does not eliminate the need for engineering rigor during system specification and design—the same rigor that has always been necessary to produce any successful system.

Modeling well is difficult. Designing well is difficult. It's possible to create a bad model. And it's possible to create a good model of a poorly designed system. Creating a good model of a well-designed system to the degree of breadth, depth, and fidelity required to satisfy the model's purpose takes time and hard work and discipline. Simply put, you can't get good things out of the model unless you do the hard work of putting them in there in the first place.

MBSE delivers its ROI when change happens—when new design decisions are made and stakeholders' needs evolve throughout the system life cycle. And change, of course, inevitably happens. Until that time, manage your stakeholders' expectations and dispel the myth of MBSE when it presents itself.

Summary

MBSE is an approach to performing systems engineering that promises to deliver a greater return on investment than the traditional document-based approach. The practice of MBSE rests on three pillars: a modeling language, a modeling method, and a modeling tool. The chapters that follow help you put that first pillar into place by teaching you SysML, the graphical modeling language that has become the de facto standard among MBSE practitioners.

Chapter 2

Overview of the Systems Modeling Language

SysML is a broad and richly expressive graphical modeling language, enabling you to visualize and communicate the essential aspects of a system's design: structure, behavior, requirements, and parametrics (mathematical models). SysML can serve as the first of the three pillars of MBSE, as discussed in Chapter 1, "Overview of Model-Based Systems Engineering."

This chapter provides a high-level view of SysML: the purpose of SysML as a whole, the purpose of each of the nine kinds of SysML diagrams, and overarching concepts that apply to all of them. This discussion provides important context for the in-depth coverage of each kind of diagram in the chapters that follow.

2.1 What SysML Is—and Isn't

SysML is one of several graphical modeling languages. The key word is *language*. SysML is a language—a medium for communicating ideas from one person to another. It has a grammar and a vocabulary just like any of the natural languages we speak (e.g., Hindi, Japanese, English). SysML is the language "spoken" by MBSE practitioners when they create system models to visualize and communicate ideas about their systems' designs to other stakeholders.

I put "spoken" in quotes because SysML is a **graphical language**. Its vocabulary consists of graphical notations that have specific meanings. For example, an arrow with a dashed line has a different meaning from an arrow with a solid line (more on these details in the chapters that follow). The key point is the purpose of SysML: visualization and communication of a system's design among stakeholders.

The grammar and notations of SysML are defined in a standards specification that is owned and published by the Object Management Group, Inc. (OMG). The OMG is a consortium of hundreds of computer industry companies, government agencies, and academic institutions that collaborate to develop a set of enterprise integration standards and promote business technology. You can find out more about the OMG on its website: www.omg.org. You will find links there also to the SysML specification document—the primary source of information about the rules of SysML.

The SysML specification attempts to define the grammar of this modeling language in a way that is precise and unambiguous, and it largely succeeds. But the text in the specification can sometimes be difficult to parse and process. The target audiences for this primary source of SysML information are the vendors of modeling tools and designers of modeling languages (and also modeling geeks who write books on the subject). In short, the SysML specification is not meant for beginners.

If that's not daunting enough, it's only half the story. As I discuss in the next section, SysML is not an independent, stand-alone language. Rather, it's a **profile**—an **extension**—of a subset of the Unified Modeling Language (UML). Therefore, if you wanted the complete definition of the grammar and vocabulary of SysML, you would need to refer also to parts of the UML specification document (also available on the OMG website).

If you're new to SysML, I recommend that you flip to Appendix A, "SysML Notation Desk Reference," to get a good first look at its various graphical notations. These notations are what actually appear on the SysML diagrams you create. These notations form the vocabulary of the systems modeling language.

So that's what SysML is: It's a modeling language. It's equally important to understand what SysML is not: It's not a modeling method. To be clear, the formal SysML specification document defines only the language itself (its grammar and vocabulary); it does not prescribe any particular modeling method.

Note

Please refer to Chapter 1 for a discussion of the distinction between modeling languages and modeling methods.

The SysML specification does not tell you, for example, at which point in the life cycle you should create, say, a use case diagram. It does not dictate that you must use an activity diagram to elaborate a use case. It does not demand that you create a set of internal block diagrams (IBDs), each focusing on a particular aspect of the system architecture. All these are methodological decisions. They fall outside the scope of the SysML specification. It is up to you and your project team to adopt and tailor a modeling method that will enable you to achieve your project's unique objectives.

2.2 Yes, SysML Is Based on UML—but You Can Start with SysML

As mentioned earlier, SysML is not an independent language; it's a profile, or extension, of UML created specifically for the systems engineering domain. UML was designed to be a standard modeling language for the software engineering domain. Systems engineers saw value in the practice of using a standard modeling language to construct models of systems, but they didn't feel that UML sufficiently captured all the concepts that are meaningful in systems engineering.

For example, UML models can contain *DataType* elements. Software engineers can use a data type (e.g., *Integer*) in a UML model to specify the type of an attribute within a class, the type of an object that can flow through an activity, and the type of a parameter within an operation. Systems engineers, however, care about other types of things that can flow—matter and energy—and not just data. The concept of *DataType* simply wasn't sufficient. Therefore, SysML introduces a new kind of model element called *ValueType,* which extends the concept of *DataType* to provide a more neutral term for the broader set of types in the systems engineering domain.

Because SysML is an extension of UML, some of the rules of SysML are actually defined in the UML specification document. This means that the SysML specification document is not a self-sufficient definition of the language. For example, if you wanted to know all the rules about using a value type in a system model, it wouldn't be sufficient to read the *ValueType* entry in the SysML specification; you would have to read the *DataType* entry in the UML specification, too. That's simply the nature of a profile as a language extension mechanism.

So do you need to go buy a UML book to learn how to create a system model?

No, you don't. This book is a sufficient primer for learning SysML so that you can get started modeling as quickly as possible. When I discuss value types in Section 3.9, "Value Types," I tell you everything you need to know about them to use them correctly and effectively in a system model. It will be transparent to you whether a specific detail comes from the UML definition of the base element, *DataType,* or from the SysML extension, *ValueType*.

2.3 SysML Diagram Overview

There are nine kinds of SysML diagrams:

- Block definition diagram (BDD)
- Internal block diagram (IBD)
- Use case diagram
- Activity diagram
- Sequence diagram
- State machine diagram
- Parametric diagram
- Package diagram
- Requirements diagram

Figure 2.1, which appears in the SysML specification v1.2, provides a good overview of the categories and relationships between the SysML diagrams. But this figure is meaningful only if you know what the lines

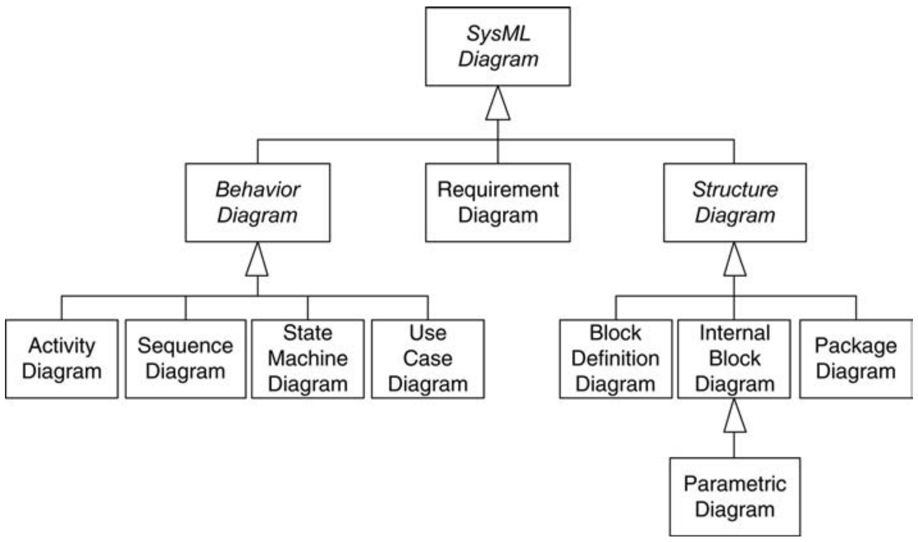

Figure 2.1 *SysML diagram taxonomy*

with the hollow, triangular arrowheads mean. They're called *generalizations*. You read them as "is a type of" in the direction of the arrowhead. (I discuss generalizations in detail in Section 3.6, "Generalizations.")

With this in mind, Figure 2.1 conveys quite a bit of information. Activity diagrams, sequence diagrams, state machine diagrams, and use case diagrams are types of **behavior** diagrams. Block definition diagrams, internal block diagrams, and package diagrams are types of **structure** diagrams. Parametric diagrams are a type of internal block diagram; therefore, a parametric diagram is transitively a type of structure diagram. Finally, **requirements** diagrams are in a category by themselves—but still a useful addition to this family of SysML diagrams.

Here's a brief summary of the purpose of each kind of diagram.

- The **block definition diagram** (BDD) is used to display elements such as blocks and value types (elements that define the types of things that can exist in an operational system) and the relationships between those elements. Common uses for a BDD include displaying system hierarchy trees and classification trees.
- The **internal block diagram** (IBD) is used to specify the internal structure of a single block. More precisely, an IBD shows the connections between the internal parts of a block and the interfaces between them.

- The **use case diagram** is used to convey the use cases that a system performs and the actors that invoke and participate in them. A use case diagram is a black-box view of the services that a system performs in collaboration with its actors.
- The **activity diagram** is used to specify a behavior, with a focus on the flow of control and the transformation of inputs into outputs through a sequence of actions. Activity diagrams are commonly used as an analysis tool to understand and express the desired behavior of a system.
- The **sequence diagram** is used to specify a behavior, with a focus on how the parts of a block interact with one another via operation calls and asynchronous signals. Sequence diagrams are commonly used as a detailed design tool to precisely specify a behavior as an input to the development stage of the life cycle. Sequence diagrams are also an excellent mechanism for specifying test cases.
- The **state machine diagram** is used to specify a behavior, with a focus on the set of states of a block and the possible transitions between those states in response to event occurrences. A state machine diagram, like a sequence diagram, is a precise specification of a block's behavior that can serve as an input to the development stage of the life cycle.
- The **parametric diagram** is used to express how one or more constraints—specifically, equations and inequalities—are bound to the properties of a system. Parametric diagrams support engineering analyses, including performance, reliability, availability, power, mass, and cost. Parametric diagrams can also be used to support trade studies of candidate physical architectures.
- The **package diagram** is used to display the way a model is organized in the form of a package containment hierarchy. A package diagram may also show the model elements that packages contain and the dependencies between packages and their contained model elements.
- The **requirements diagram** is used to display text-based requirements, the relationships between requirements (containment, derive requirement, and copy), and the relationships between requirements and the other model elements that satisfy, verify, and refine them.

2.4 General Diagram Concepts

You should be aware of a few overarching concepts about SysML diagrams before you delve into the details of the specific types. A sample SysML diagram is shown in Figure 2.2.

Each diagram has a frame, a contents area (colloquially called the "canvas"), and a header. The diagram **frame** is the outer rectangle. The **contents area** is the region inside the frame where model elements and relationships can be displayed. The **header** is in the upper-left corner of the diagram, shown in Figure 2.2 with its lower-right corner cut off.

In SysML (unlike in UML), the frame must be displayed. With that said, some figures in this book show model elements and relationships without an enclosing frame. I do that to hide inconsequential information and focus your attention on particular notations. Officially, though, the frame is mandatory.

One of the most important diagram concepts is the format of the header information. The header commonly contains four pieces of information:

- Diagram kind
- Model element type
- Model element name
- Diagram name

The format of that information is shown in the header in Figure 2.3.

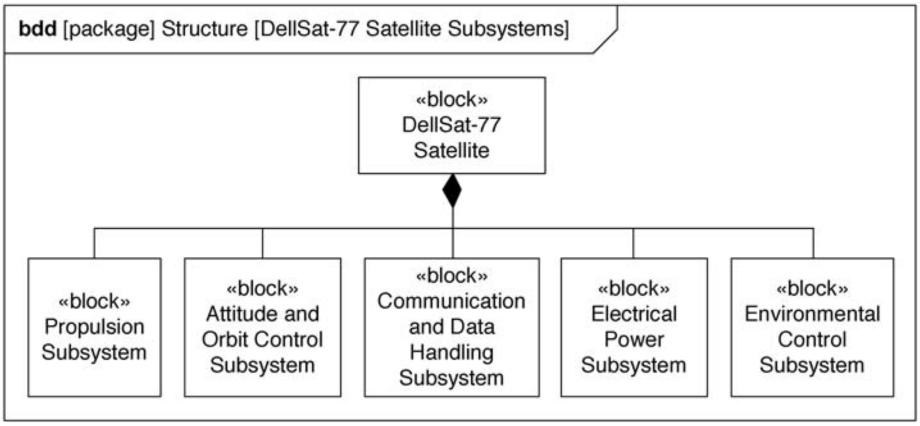
bdd [package] Structure [DellSat-77 Satellite Subsystems]
«block» DellSat-77 Satellite
«block» Propulsion Subsystem
«block» Attitude and Orbit Control Subsystem
«block» Communication and Data Handling Subsystem
«block» Electrical Power Subsystem
«block» Environmental Control Subsystem

Figure 2.2 *Sample SysML diagram*

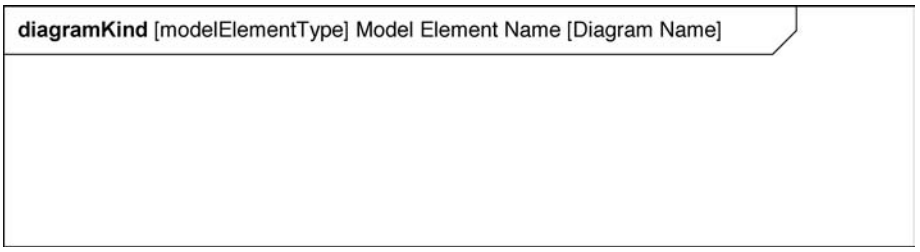
diagramKind [modelElementType] Model Element Name [Diagram Name]

Figure 2.3 *Diagram header format*

I begin with two intuitive pieces of information: diagram kind and diagram name. The **diagram kind** is shown as its SysML-defined abbreviation:

- **bdd** = block definition diagram
- **ibd** = internal block diagram
- **uc** = use case diagram
- **act** = activity diagram
- **sd** = sequence diagram
- **stm** = state machine diagram
- **par** = parametric diagram
- **req** = requirements diagram
- **pkg** = package diagram

Based on this, you can conclude that the diagram in Figure 2.2 is a block definition diagram. (Chapter 3, "Block Definition Diagrams," discusses in detail the kinds of elements that can appear on a BDD.)

The **diagram name** can be anything you want it to be. I advise you to choose a diagram name that conveys which aspect of the model is in focus on that diagram. For example, the name of the diagram in Figure 2.2 is "DellSat-77 Satellite Subsystems." This diagram name indicates that the focus of the diagram is the set of subsystems that make up the satellite system. The model certainly contains other information about the satellite system, but that information is not the focus of this diagram.

The next two pieces of information in the header are the model element type and the model element name. To understand what these refer to, you first need to know another essential concept about SysML diagrams: Each diagram you create represents an element that you've defined somewhere in your system model. More precisely, the diagram frame represents an element in the model. And *that* model element is the element whose type and name appear in the diagram header.

In Figure 2.2, the model element type is "package," and the model element name is "Structure." This conveys that the frame of this BDD represents the *Structure* package that exists somewhere in the system model hierarchy.

Requiring each diagram to represent a model element may seem like a strict and unnecessary constraint, but in the words of Frederick Brooks, Jr., in *The Design of Design* (Boston: Addison-Wesley, 2010), "Constraints are friends" (p. 127). The connection between a diagram and a model element was a deliberate and brilliant decision on the part of the SysML authors. The reason is conveyed by the next key concept of SysML diagrams: The model element represented by the diagram defines the **namespace**—the container element within the model hierarchy—for the other elements shown on the diagram. Simply put, the model element type and model element name shown in the diagram header indicate where the elements on the diagram can be found within the model.

The diagram header in Figure 2.2 tells us that the six blocks shown in the contents area are contained in (i.e., nested under) the *Structure* package in the model hierarchy. This gives us a sense of how elements are partitioned in the model and aids us in navigating it.

The model element may be a structural element (e.g., a package or block), or it may be a behavioral element (e.g., an activity, interaction, or state machine). The type of model element that a diagram can represent depends on the kind of diagram you're creating. The pairings are shown in Table 2.1.

Recall from Chapter 1 the distinction between a system model (as an engineering artifact in its own right) and the set of diagrams you create (which are views of the underlying model). This idea is so important that here I rephrase it more formally and give it a name: the fundamental precept of model-based engineering. Your assignment is to assume full lotus position and repeat the following mantra until you enter a deep meditative state:

> ***A diagram of the model is never the model itself; it is merely one view of the model.***

This idea is a paradigm shift for engineers who have only ever sketched designs using paper, whiteboards, or diagramming tools. And it's an idea best explained via metaphor: The model is a mountain,

Table 2.1 *Allowable Model Element Types for Each Diagram Kind*

Diagram Kind	**Allowable Model Element Types**
Block definition diagram	package, model, modelLibrary, view, block, constraintBlock
Internal block diagram	block
Use case diagram	package, model, modelLibrary, view
Activity diagram	activity
Sequence diagram	interaction
State machine diagram	stateMachine
Parametric diagram	block, constraintBlock
Requirement diagram	package, model, modelLibrary, view, requirement
Package diagram	package, model, modelLibrary, view, profile

and a diagram is a picture of the mountain. The mountain exists whether or not anyone takes a picture of it. If a man comes along and takes a picture from the north side, he creates one view of the mountain that shows some of its features but not others. If a woman takes a picture from the west side, she creates a second view of the mountain that shows a different—possibly overlapping—set of features.

Each of the two views focuses on a different aspect of the whole. But at all times the pictures are only views of the mountain and not the mountain itself. The mountain and its features will continue to exist even if the photographers later crop out certain details from the final pictures . . . and even if they destroy the pictures.

This metaphor fails in one respect: You can add new features and modify or delete existing features from the model at any time. When you modify an existing feature, the changes are instantly reflected on all diagrams that show the feature. When you delete a feature from the model, it instantly vanishes from all diagrams that showed that feature. Clearly the pictures in your photo album don't offer the same capability.

Earlier in this section I mentioned the practice of eliding inconsequential information on a given diagram. No diagram should attempt to convey every detail; the diagram would be unreadable. You should instead decide what you want the focus of a given diagram to be and then elide all model information that is not within that focus. This idea leads to the corollary to the fundamental precept of model-based engineering:

> ***You cannot conclude that a feature doesn't exist from its absence on a diagram; it may be shown on another diagram of the model or on no diagram at all.***

Summary

SysML is a richly expressive graphical modeling language that you can use to visualize the structure, behavior, requirements, and parametrics of a system and communicate that information to others. SysML defines nine kinds of diagrams that you can use to convey all this system design information; each kind serves a specific purpose and conveys specific information about an aspect of a system.

The chapters that follow provide detailed coverage of these diagrams. You will learn the various kinds of SysML model elements—and the relationships among them—that can appear on each kind of diagram. I include discussions of the rules of SysML that you will need to know to build your system model correctly and ensure effective communication with your stakeholders.